Distributed Search

Suresh Podeti

6 min read · Sep 28, 2023

Photo by Daniel Lerman on Unsplash

Introduction

Nowadays, we see a search bar on almost every website. A search bar enables us to quickly find what we're looking for.

Let's take another example. Imagine if YouTube didn't provide us with a search bar. How would we find a specific video among the millions of videos that have been

posted on YouTube over the years? Users find it challenging to find what they're looking for simply by scrolling around.

Behind every search bar, there is a search system.

Requirements

- **Availability:** The system should be highly available to the users.
- **Scalability:** The system should have the ability to scale with the increasing amount of data. In other words, it should be able to index a large amount of data.
- **Fast search on big data:** The user should get the results quickly, no matter how much content they are searching.

Core concepts

Inverted Indexing

Indexing — is the organization and manipulation of data that's done to **facilitate fast** and accurate **information** retrieval.

Inverted index — is a **HashMap-like** data structure that employs a **document-term matrix.** Instead of storing the complete document as it is, it splits the documents into individual words. After this, the **document-term matrix** identifies unique words and discards frequently occurring words like "to," "they," "the," "is," and so on.

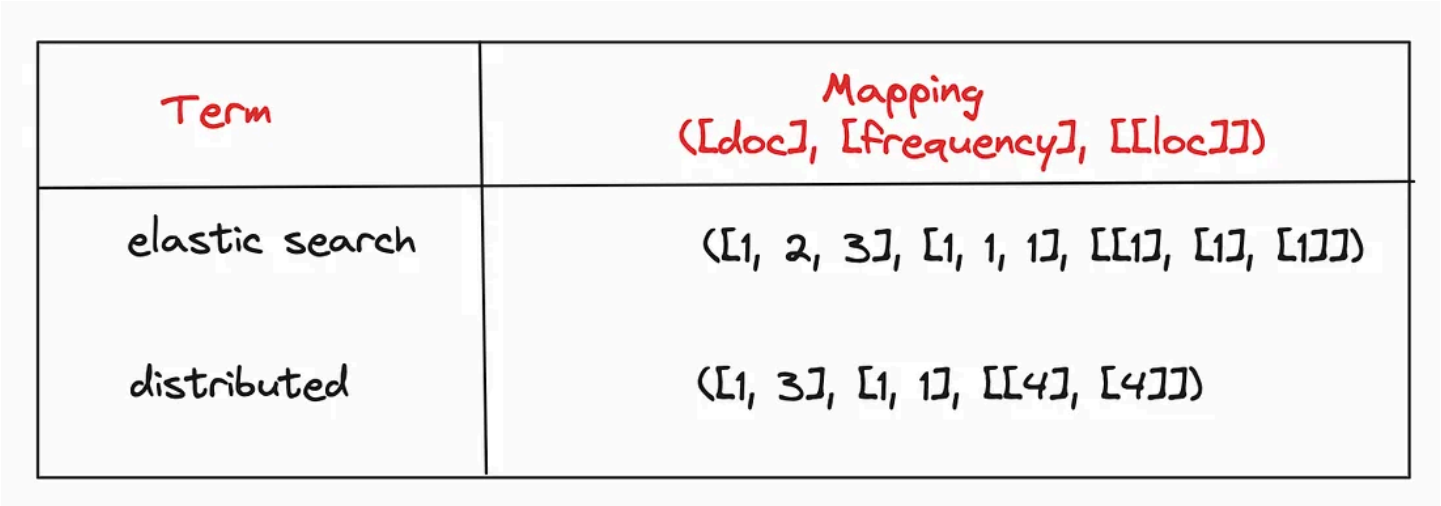

Fig 1.0: Inverted Index

Each entry in the "Mapping" column consists of three lists:

- A list of **documents** in which the term appeared.

- A list that **counts the frequency** with which the term appears in each document.
- A **two-dimensional list that pinpoints the position** of the term in each document. A term can appear multiple times in a single document, which is why a two-dimensional list is used.

For each extracted term, we either add a new row in the inverted index or update an existing one if that term already has an entry in the inverted index. Similarly, for deleting a document, we conduct processing to find the entries in the inverted index for the deleted document's terms and update the inverted index accordingly.

Design

An **inverted index needs to be loaded into the main memory** when adding a document or running a search query. A large portion of the inverted index must fit into the **RAM** of the machine for efficiency.

This means we have to load lot of data into the RAM. It's impractical and inefficient to increase the resources of a single machine for indexing a billion pages instead of **shifting to a distributed system** and utilizing the **power of parallelization.**

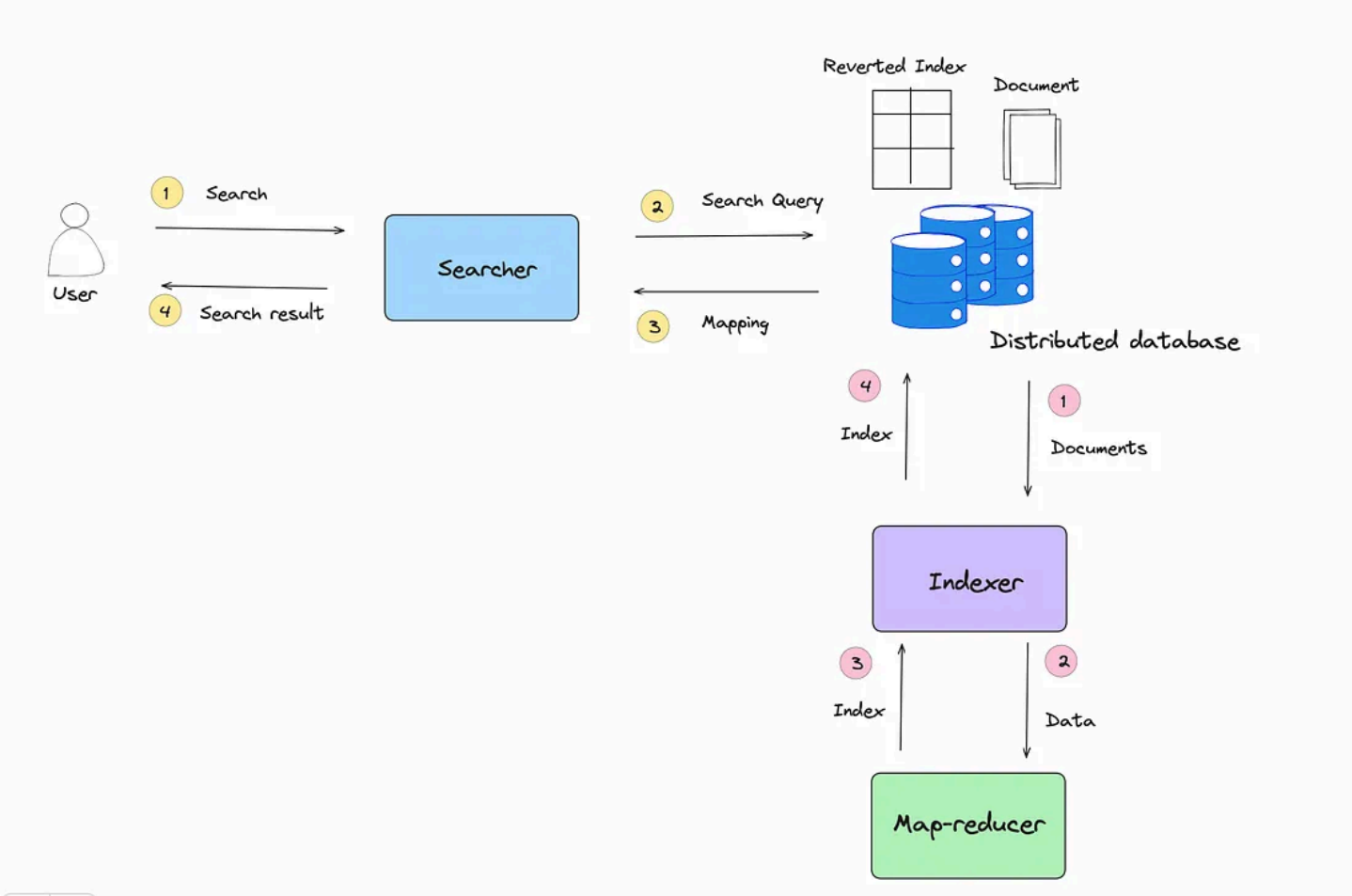

Fig 2.0: High-level design of a distributed search system

- The **indexer** fetches the documents from a distributed storage and indexes these documents using **MapReduce**, which runs on a distributed cluster of commodity machines. The indexer uses a **distributed data processing system** like MapReduce for parallel and distributed index construction. The constructed index table is stored in the distributed storage.
- The **distributed storage** is used to store the documents and the index.
- The **user** enters the search string that contains multiple words in the search bar.
- The **searcher** parses the search string, searches for the mappings from the index that are stored in the distributed storage, and returns the most matched results to the user.

Data partitioning

We use **numerous small nodes** for indexing to achieve cost efficiency. This process requires us to partition or **split the input data** (documents) **among these nodes.**

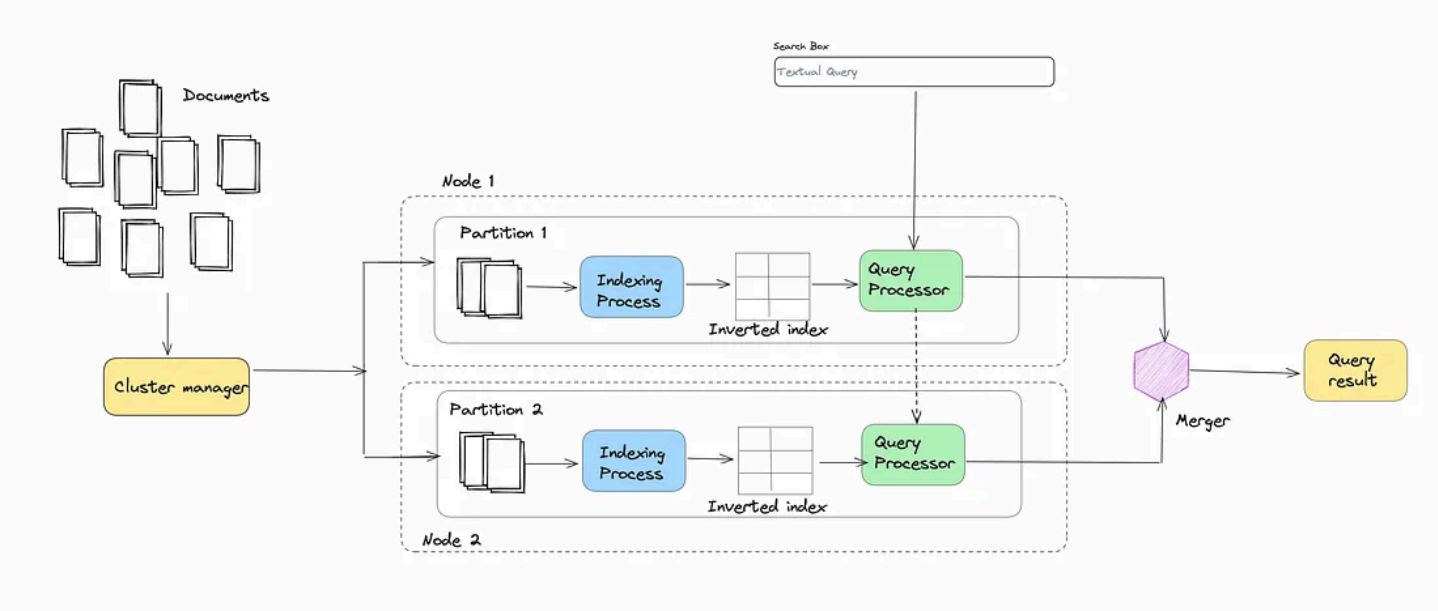

Fig 3.0: Distributed indexing and searching in a parallel fashion on multiple nodes in a cluster of commodity machines

Indexing

- The **cluster manager** splits the input document set into *N* number of partitions, where *N* is equal to two in the illustration above. The size of each partition is decided by the cluster manager given the size of the data, the computation,

memory limits, and the number of nodes in the cluster. All the nodes may not be available for various reasons. The cluster manager monitors the health of each node through **periodic heartbeats**. To assign a document to one of the *N* partitions, a **hashing function** can be utilized.

- After making partitions, the cluster manager runs indexing algorithms for all the *N* partitions simultaneously on the *N* number of nodes in a cluster. Each indexing process produces a tiny inverted index, which is stored on the node's local storage. In this way, we produce *N* tiny inverted indices rather than one large inverted index.

Searching

- In the search phase, when a user query comes in, we run parallel searches on each tiny inverted index stored on the nodes' local storage generating N queries.
- The search result from each inverted tiny index is a mapping list against the queried term (we assume a single word/term user query). The **merger** aggregates these mapping lists.
- After aggregating the mapping lists, the merger sorts the list of documents from the aggregated mapping list based on the frequency of the term in each document.
- The sorted list of documents is returned to the user as a search result. The documents are shown in sorted (ascending) order to the user.

Replication

We make replicas of the indexing nodes that produce inverted indices for the assigned partitions.

Generally, a replication factor of three is enough. A replication factor of three means three nodes host the same partition and produce the index. **One of the three nodes** becomes the **primary node**, while the other **two are replicas**. Same partition will be forwarded to all three replicas. We assume that each replica will compute the index individually, which leads to inefficient usage of resources. **Rather than recomputing** the index on each replica, we **compute the inverted index on the primary node only**. Next, we communicate the inverted index (binary blob/file) to the replicas. The key benefit of this approach is that it avoids using the duplicated amount of CPU and memory for indexing on replicas.

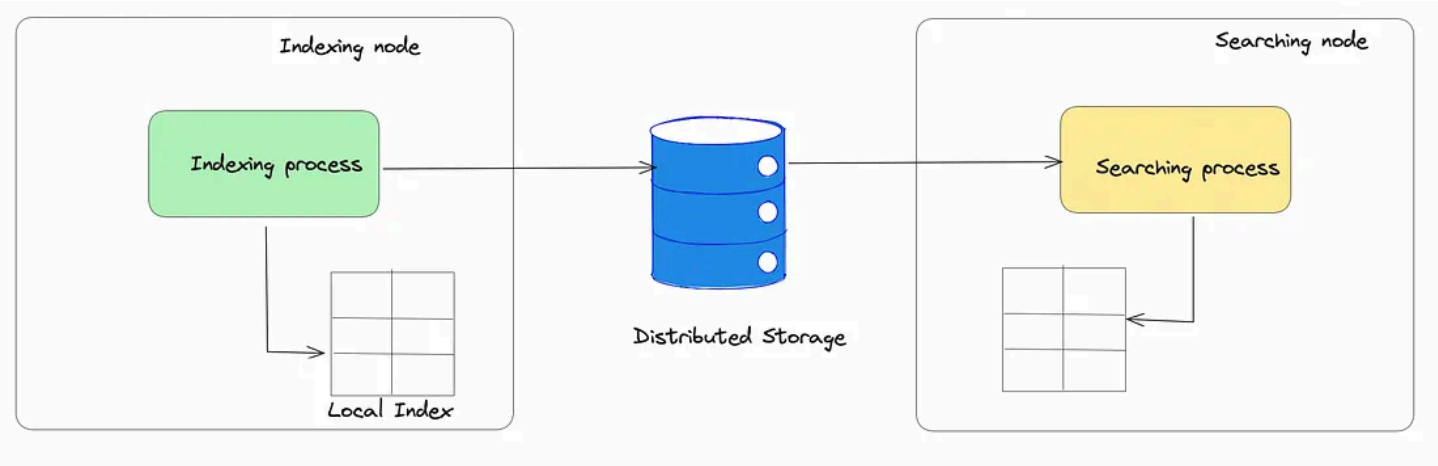

Fig 4.0: The indices produced by the indexing nodes are stored on the distributed storage, and the nodes involved in the search reads indices from the distributed storage to produce a result for the user's query

There is a strong **separation between indexing** and **search** without the negative consequence of indexing latency. Because of this **isolation, indexing wouldn't affect search scalability and vice versa**. Also, instead of recomputing the index on the replica nodes, which wastes resources, we can just replicate the index files.

In the case of a hardware failure, a new searcher or indexer machine is added, and a copy of the data is retrieved from the distributed storage.

Evaluation

Availability

Data is replicated across multiple regions in distributed storage, making cross-region deployment for indexing and search easier. So, if a failure occurs in one place, we can process the requests from another cluster.

The indexing is performed offline, not on the user's critical path. We don't need to replicate the indexing operations synchronously. It is unnecessary to respond to the user search queries with the latest data that has just been added to the index. So, we don't have to wait for the replication of the new index to respond to the search queries. This makes the search available to the users.

Scalability

Partitioning is an essential component of search systems to scale. When we increase the number of partitions and add more nodes to the indexing and search clusters, we can scale in terms of data indexing and querying.

The strong isolation of indexing and search processes help indexing and search scale independently and dynamically.

Fast search on big data

We utilized a number of nodes, each of which performs search queries in parallel on smaller inverted indices. The result from each search node is then merged and returned to the user.

Distributed Search | Software Architecture | Software Architect | Distributed Systems | System Design Interview

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium